



Chief Reader Report on Student Responses: 2025 AP[®] Computer Science A Free-Response Questions

• Number of Students Scored	93,906		
• Number of Readers	479		
• Score Distribution	Exam Score	N	%At
	5	23,909	25.5
	4	20,433	21.8
	3	18,558	19.8
	2	10,236	10.9
	1	20,770	22.1
• Global Mean	3.18		

The following comments on the 2025 free-response questions for AP[®] Computer Science A were written by the Chief Reader, Don Blaheta, Associate Professor of Computer Science at Longwood University. They give an overview of each free-response question and of how students performed on the question, including typical student errors. General comments regarding the skills and content that students frequently have the most problems with are included. Some suggestions for improving student preparation in these areas are also provided. Teachers are encouraged to attend a College Board workshop to learn strategies for improving student performance in specific areas.

Notes about the 2025 exam and the new CED

The exams administered in May 2025 represented a change from prior practice, in that all exams were administered digitally, through the Bluebook software provided by College Board. This was an advance from the previously announced timeline, but the software itself had been in development for many years and the roll-out of the new medium generally went smoothly. The exam format itself, meaning the type and content of the questions, was not affected by this change, and continued to be exactly as specified in the Course and Exam Description (CED) of 2019.

The new CED, released in February 2025, will be in effect for courses taught in the 2025-26 academic year, and for the exams administered in May 2026.

What were the effects of transitioning to the digital exam administration?

Obviously and predictably, Readers no longer had to read handwritten responses and so handwriting was never an issue. However, as the program code of the digital responses had merely been typed into a text editor, and never compiled, we immediately saw that some of the rules we had understood as “allowing for handwriting” were in fact “allowing for quickly-written and un-compiled code”. We also saw an immediate need to codify, at least for now, how to interpret certain kinds of incorrect programs, in order to have consistency in the scoring.

The second page of each Scoring Guidelines document this year was labeled “2025 Digital Decision Rules”. Take note that we will be revisiting the decision rules, so teachers and students should not assume that any particular missing, extra, or misplaced element will be ignored. (They should *never* assume that: students should always attempt to write the most correct solutions they can).

A few elements from “page 2” deserve additional note. First, the notes about non-ASCII characters were truly a one-off issue for the 2025 exam; we were able to identify the problem and it will not be an issue in the future.

Second, the most unexpected effect of the digital administration was the number of submitted responses that were mostly or entirely without indentation. Java does not require indentation and it is certainly possible to write a perfect, compilable, correct Java class or method with no indentation at all (and such a response would receive full credit). The difficulty arose when brackets (or, sometimes, parentheses) were missing in such a response. We have had decades-old decision rules about handling missing brackets when “indentation clearly conveys intent”, and they served us well enough in the handwritten era, when “left-justified” responses were exceedingly rare. Expect the page 2 decision rules to evolve a little, but some phrase like “where indentation clearly conveys intent” is likely to remain. Keep in mind that responses are a communication from one human (the student) to another (the Reader): indentation remains one of the widest and best-understood ways to convey *intended* structure of a program to other humans, so indentation is a good extra communication channel for us to use when mistyped or omitted punctuation would lead the compiler astray.

How do the notes in this report apply to future exams designed under the 2025-26 Course and Exam Description?

With the exception of the small number of removed topics (notably, inheritance) and a small number of new topics (mostly focused on data processing), the topic coverage of the 2025 CED is extremely similar to that of the 2019 CED and earlier. Inheritance was not assessed in the FRQ section of the exam form being analyzed in this report, so all notes about content, strengths, misconceptions, etc., will apply equally well to courses being taught next year.

The notes in this report are grouped in a way that is keyed to the 2019 CED, but analysis and feedback appropriate to [old] “Skill 3.C Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements”, for instance, will apply perfectly well to improve [new] “Skill 2.A Write program code to implement an algorithm”. However, some recommendations are mapped to both old and new topic and unit numbers for convenience in looking up resources.

Question 1

Task: Methods and Control

Topic: Dog walk scheduling

	Max Points:	Mean Score:
Point 1: Call methods on object	1	0.27
Point 2: Compare values	1	0.70
Point 3: Calculate and update (algorithm)	1	0.38
Point 4: Return calculation	1	0.67
Point 5: Iterate numeric range	1	0.58
Point 6: Call method with parameter	1	0.59
Point 7: Multiply by constant	1	0.63
Point 8: Conditionally add bonus	1	0.40
Point 9: Accumulate total (algorithm)	1	0.32
Overall Mean Score:	4.54	

What were the responses to this question expected to demonstrate?

This question tested the student's ability to:

- Write program code to call methods (Skill 3.A).
- Write program code to define a new type by creating a class (Skill 3.B).
- Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements (Skill 3.C).

This question assesses the ability to call instance methods on an object of another class and on the current object, use comparisons to select different code segments, iterate through a selected period of time, accumulate a total, and return that accumulated amount.

In part (a) students were asked to write the `walkDogs` method in the `DogWalker` class. This method should determine the number of dogs a dog-walking employee could walk in a given hour based upon the maximum number that dog walker could walk and the number available in that hour. Should there be more dogs available than the walker could walk, then they should walk `maxDogs`. In the other case, the walker should walk the remaining number of available dogs. In either case, after selecting the correct number of dogs to walk, the pool of available dogs should be updated and the number of dogs walked should be returned.

In part (b) students were asked to write the `dogWalkShift` method in the `DogWalker` class. This method should determine the pay for a dog walking shift based on the number of dogs walked and when they were walked. For each hour in the shift, students were asked to calculate the base pay by multiplying the number of dogs walked by 5, and to sometimes add an additional bonus based on certain conditions. The total amount earned is the sum of the calculated pay for each hour of the shift.

How well did the responses address the course content related to this question? How well did the responses integrate the skill(s) required on this question?

Write program code to call methods (Skill 3.A).

This question required calls to a non-void method in another class, a void method in another class, and a non-void method in the same class. Most responses made some method calls, but not always correctly. In

particular, the methods being written in this question are part of the `DogWalker` class, but both of the method calls in part (a) were to methods in the `DogWalkCompany` class, and thus needed to be called using the instance variable `company`—but the overwhelming majority of responses omitted the `company` object from the method calls. By contrast, in part (b), students needed to call `walkDogs(hour)`, a method within the same class, to determine how many dogs were walked in a specific hour, and most responses were at least syntactically successful on their `walkDogs` calls.

Among responses that called the methods in part (a) incorrectly, most did not call the methods on any object or class, as mentioned. Several others that wrote this incorrectly attempted to call these non-static methods on the `DogWalkCompany` class. Occasionally, responses would attempt to return the result of a `void` method call. There are other Java methods, outside of the subset described in the Course and Exam Description (CED), that provide functionality that is useful for comparing two integer values (such as `Math.min`), and a small number of responses used them successfully, earning credit. The exam never requires the use of methods outside that subset; but responses that use them incorrectly will not earn the associated points.

The most common method-calling error in part (b) was to omit the parameter for the `walkDogs` method.

Write program code to define a new type by creating a class (Skill 3.B).

Although this skill is primarily tested in Q2 (Class Design), one aspect of this skill involves understanding that providing the result of a computation is done through a `return` statement, and that a method must return a value of the correct type and must return something in all cases. Both methods were expected to return a value, and doing so was formally assessed in part (a). More than two-thirds of responses returned a calculated value in all cases in part (a). Of the ones that did not, some attempted to return the result of a `void` method or a constant, but many failed to return a value at all.

Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements (Skill 3.C).

This question primarily tested this skill. Both parts of the question involved using conditional logic to control the operations containing expressions, and part (b) required iteration to incorporate these skills into a larger algorithm. The first part of the question required using a conditional statement to determine the correct integer value to pass into the call to `updateDogs`. The second part required a summation algorithm that accumulated the pay for a range of hours with the possibility, conditionally, of bonus pay for each hour.

In part (a), responses needed to determine the number of dogs that were available for a given hour. Responses needed to compare a value retrieved from the call to `numAvailableDogs` to the private instance variable `maxDogs`. The overwhelming majority of responses made this comparison correctly to start their algorithm. Responses that did not earn credit for the comparison point either did not attempt a comparison, or compared `maxDogs` to a constant value, such as `0`.

A common error in the algorithm was in calling `numAvailableDogs` *after* a call to `updateDogs`, typically in an attempt to determine the number of dogs walked. However, the `updateDogs` method changes the number of dogs available for a walk; therefore any subsequent call to `numAvailableDogs` will in general return a different value.

In part (b), responses needed to loop through all the hours from `startHour` to `endHour`, inclusive, calculate the base pay for each hour, determine if bonus pay was earned for each hour, and accumulate both types of pay in a variable representing the total pay for a shift. Responses typically iterated through the correct

hours with a loop variable initialized with `startHour` and incremented through `endHour`. Some correct variations on this looped from `0` over the number of hours in the shift and added `startHour` to the looping variable when calling `walkDogs`. Incorrect variations of this approach either miscalculated the number of hours in the shift or failed to offset their looping variable by `startHour`. The majority of responses calculated the base pay for an hour correctly by multiplying the number of dogs walked by 5. The bonus pay determination, however, required careful reasoning through conditional statements and proved challenging for many. Incorrect responses often had difficulty determining if the tested hour was a “peak hour” and instead checked whether the entire shift was within the peak hour range. A significant number of responses attempted to write the inequality expressions algebraically, `9 <= i <= 17`, instead of creating a compound conditional expression.

Many responses failed to earn the final point, for accumulating total pay. Sometimes this was for a straightforward reason, such as failing to initialize the accumulation variable or assigning to the variable instead of accumulating. However, a more subtle error in some responses was calling the method `walkDogs` multiple times within the loop without recognizing that the side effects from this method call could potentially alter the value received on subsequent calls. Because multiple calls to `walkDogs` could result in incorrect accumulation, this error was included with the other accumulation errors and assessed in Point 9.

What common student misconceptions or gaps in knowledge were seen in the responses to this question?

<i>Common Misconceptions/Knowledge Gaps</i>	<i>Responses that Demonstrate Understanding</i>
<i>Write program code to call methods.</i>	
Some responses called the instance methods on the class or without a reference to the instance. <pre>DogWalkCompany.numAvailableDogs(hour); DogWalkCompany.updateDogs(hour, dogs);</pre> OR <pre>numAvailableDogs(hour); updateDogs(hour, dogs);</pre>	<pre>company.numAvailableDogs(hour); company.updateDogs(hour, dogs);</pre>
Some responses incorrectly called the instance methods without a parameter. <pre>company.numAvailableDogs();</pre>	<pre>company.numAvailableDogs(hour);</pre>
Some responses incorrectly tried to use a (nonexistent) return value from a void method. <pre>return company.updateDogs(hour, maxDogs);</pre>	<pre>company.updateDogs(hour, maxDogs); return maxDogs;</pre>

Some responses called the <code>walkDogs</code> method without a parameter. <pre>walkDogs();</pre>	<pre>walkDogs(hour);</pre>
Some responses incorrectly called the <code>walkDogs</code> method on the class. <pre>DogWalker.walkDogs(hour);</pre>	<pre>walkDogs(hour);</pre>
Some responses incorrectly called the <code>walkDogs</code> method outside the loop, with an argument that thus had no value. <pre>int dogs = walkDogs(hour); for (int hour = ...)</pre>	<pre>for (int hour = ...) { int dogs = walkDogs(hour); ... }</pre>
<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to define a new type by creating a class.</i>	<i>Responses that Demonstrate Understanding</i>
Some responses incorrectly printed to output in addition to or instead of returning a value. <pre>System.out.println("We walked " + dogs + " dogs"); return dogs;</pre>	<pre>return dogs;</pre>
Some responses returned a concatenated <code>String</code> instead of an integer. <pre>return "We walked " + dogs + " dogs";</pre>	<pre>return dogs;</pre>
<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements.</i>	<i>Responses that Demonstrate Understanding</i>
Some responses incorrectly accessed the instance variable by calling it on the class. <pre>if (dogs <= DogWalker.maxDogs) {...}</pre>	<pre>if (dogs <= maxDogs) {...}</pre>

Some responses incorrectly called <code>numAvailableDogs</code> after the <code>updateDogs</code> method had been called. <pre>company.updateDogs(hour, company.numAvailableDogs(hour)); return company.numAvailableDogs(hour);</pre>	<pre>int dogs = company.numAvailableDogs(hour); company.updateDogs(hour, dogs); return dogs;</pre>
Some responses incorrectly identified all the possible cases when making a comparison to <code>maxDogs</code>. <pre>int dogs = company.numAvailableDogs(hour); if (dogs > maxDogs) {...} else if (dogs < maxDogs) {...}</pre>	<pre>int dogs = company.numAvailableDogs(hour); if (dogs > maxDogs) {...} else {...}</pre> OR <pre>int dogs = company.numAvailableDogs(hour); if (dogs >= maxDogs) {...} else if (dogs < maxDogs) {...}</pre>
Some responses incorrectly identified the number of dogs to walk. <pre>if (dogs > maxDogs) { company.updateDogs(hour, dogs); return dogs; }</pre>	<pre>if (dogs > maxDogs) { company.updateDogs(hour, maxDogs); return maxDogs; }</pre>
Some responses incorrectly calculated the base pay by using hours worked instead of dogs walked. <pre>int shift = endHour - startHour; pay += shift * 5;</pre>	<pre>pay += dogs * 5;</pre>
Some responses incorrectly counted the bonus twice when both conditions were true. <pre>if (dogs == maxDogs) { pay += ...} if (x >= 9 && x <= 17) { pay += ...}</pre>	<pre>if (dogs == maxDogs) { pay += ...} else if (x >= 9 && x <= 17) { pay += ...}</pre> OR <pre>if (dogs == maxDogs (x >= 9 && x <= 17)) { pay += ...}</pre>

Some responses used incorrect logic operators in their conditional statement. <pre>if (dogs == maxDogs x >= 9 x <= 17)</pre> OR <pre>if (dogs == maxDogs && x >= 9 && x <= 17)</pre> OR <pre>if (dogs == maxDogs && x >= 9 x <= 17)</pre>	<pre>if (dogs == maxDogs x >= 9 && x <= 17)</pre>
Some responses compared the <code>startHour</code> and/or <code>endHour</code> instead of the loop variable in their conditional statement. <pre>for (int x = startHour; ...) { if (startHour >= 9 && endHour <= 17) }</pre>	<pre>for (int x = startHour; ...) { if (x >= 9 && x <= 17) }</pre>
Some responses failed to initialize the accumulator. <pre>int total; for (...) { total += pay; total += bonus; }</pre>	<pre>int total = 0; for (...) { total += pay; total += bonus; }</pre>
Some responses called the <code>walkDogs</code> method multiple times per loop, causing an incorrect accumulation. <pre>for (...) { pay += walkDogs(x) * 5; if (walkDogs(x) == maxDogs) {...} }</pre>	<pre>for (...) { int dogs = walkDogs(x); pay += dogs * 5; if (dogs == maxDogs) {...} }</pre>

Some responses had a `return` statement inside the loop, causing an early return and an incorrect accumulated result.

```
for (...)  
{  
    ...  
    return total;  
}
```

```
for (...)  
{  
    ...  
}  
return total;
```

Based on your experience at the AP® Reading with student responses, what advice would you offer teachers to help them improve student performance on the exam?

- Practice calling methods in a variety of situations: on an object in an instance variable, on an object in a parameter variable, on an object in a local variable, on the current object, on a different class.
- Practice calling methods with a variety of parameters.
- Practice conditional statements in the context of loops.
- Practice common arithmetic algorithms, such as summation.
- Practice compound Boolean statements that utilize logical operators (`&&` and `||`).
- Practice using concrete examples involving edge case scenarios (e.g., “what if there are not `maxDogs` left to walk”, “what if there are exactly `maxDogs` left to walk”) to demonstrate that calling `numAvailableDogs` changes with each call to `updateDogs` and `walkDogs`.
- Practice looping through known quantities that don’t start at zero (e.g., “loop through all class periods after lunch”, “loop through hours in a school day”) to ensure that all values are accessed.
- If students are inclined to use parts of Java outside of the subset described in the CED (for instance, the `Math.min` method), remind them that unless they are very sure of what they are doing it is safer to only use the Java classes, methods, and features covered in the course. If you teach some of the non-CED parts of Java (as many teachers do very successfully), make sure the students also are comfortable with the testable classes and methods.

What resources would you recommend to teachers to better prepare their students for the content and skill(s) required on this question?

- Progress checks from units 1 and 2 (formerly units 2, 3, and 4) would be helpful to scaffold students’ understanding for the Methods and Control free-response questions.
- The following Topic Questions can be found in AP Classroom to support this Methods and Control free-response question:
 - Write program code to create objects of a class and call methods. Topic 1.9 (formerly 2.3, 2.4, and 2.5).
 - Write program code to define a new type by creating a class. Topic 3.5 (formerly 2.5).
 - Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements. Topics 2.2, 2.3, 2.4, 2.7, and 2.8 (formerly 3.1, 3.2, 3.3, 3.4, 4.1, and 4.2).

Question 2

Task: Class Design

Topic: Signatures

	Max Points:	Mean Score:
Point 1: Class header	1	0.77
Point 2: Instance variables	1	0.61
Point 3: Constructor header	1	0.70
Point 4: Method headers	1	0.66
Point 5: Empty string comparison	1	0.60
Point 6: Construct string (algorithm)	1	0.63
Point 7: Call <code>String</code> methods	1	0.61
Point 8: Distinguish three cases	1	0.34
Point 9: Return revised string (algorithm)	1	0.32
Overall Mean Score: 5.24		

What were the responses to this question expected to demonstrate?

This question tested the student's ability to:

- Write program code to create objects of a class and call methods (Skill 3.A).
- Write program code to define a new type by creating a class (Skill 3.B).
- Write program code to satisfy method specifications using expressions and conditional statements (Skill 3.C).

This question assessed the ability to design a complete class named `SignedText` that models a signature constructed from a first and last name. The class includes a constructor that takes two `String` parameters (first name and last name) and two methods: `getSignature` and `addSignature`. The `getSignature` method returns a formatted signature string based on the first name and last name, following specific rules about when to include the first initial and how to format the signature. The `addSignature` method takes a `String` parameter and returns a possibly revised version of that string, adding or repositioning the signature according to the rules specified.

Students were expected to demonstrate understanding of class and method header syntax, proper declaration and initialization of private instance variables, and correct use of string manipulation, conditional logic, and string comparison to implement the specified behaviors. The problem requires careful handling of empty strings, string concatenation, and substring operations.

How well did the responses address the course content related to this question? How well did the responses integrate the skill(s) required on this question?

Write program code to create objects of a class and call methods (Skill 3.A).

Students were expected to demonstrate skills comparing strings to determine whether the signature string was present at the beginning or end of the `addSignature` parameter `String`. This requires using the `equals` method (or equivalent string comparisons) rather than the `==` operator to compare string contents accurately. Students also demonstrated their ability to use `String` methods such as `substring` and `length` with correct syntax and arguments.

In addition, a typical strategy requires invoking the `getSignature` method within the `addSignature` method to obtain the signature string needed for comparison and string manipulation. Responses using this strategy were generally successful in making the method call.

Write program code to define a new type by creating a class (Skill 3.B).

The headers for the class, the constructor, and both required methods (`getSignature` and `addSignature`) were tightly constrained by the prompt and the examples, and a large majority of responses wrote them correctly—these were the most widely-earned points in the entire FRQ section, as they often are. This included the use of the `public` keyword, correct class and method names, appropriate return types, and correct parameter lists. No single error was especially common in this area, but some responses omitted one or more required lines entirely, or combined the class header with the constructor header by mistakenly adding parentheses or parameters to the class header itself.

Declaring and initializing instance variables was somewhat more challenging, although a majority of responses were still successful. Most responses correctly declared private instance variables for the first and last names inside the class but outside any method or constructor. A few responses omitted the `private` access modifier, which violates the CED’s mandate that “attributes should be kept internal to the class” and thus does not earn credit for the instance variable point. Several responses incorrectly attempted to declare instance variables inside the constructor, which is not valid.

Most responses correctly initialized the instance variables inside the constructor using the constructor parameters. Some responses declared and initialized instance variables on the same line inside the class (outside the constructor), which would work if initializing with literals or constants, but does not work when the initial value depends on constructor parameters such as, in this case, the first and last names. Some responses correctly used the keyword `this` to distinguish between instance variables and constructor parameters with the same name, but others used identical names without `this`, which prevented proper initialization.

A large majority of responses accessed instance variables appropriately within the methods. However, a few responses incorrectly attempted to use the class name to access instance variables (e.g., `SignedText.firstName`), which is invalid for instance variables. Some responses declared local variables inside methods and incorrectly attempted to use them as persistent instance data, which led to errors in maintaining object’s state across method calls.

Write program code to satisfy method specifications using expressions and conditional statements (Skill 3.C).

As is often the case with class design questions, the algorithm implementation was not entirely contained within the method bodies but was also affected by design choices involving the types and initial values of instance variables. In particular, any correct response needed to store the first and last names as strings and process them appropriately, although at least some of the processing could be done either in advance (in the constructor) or just-in-time (in the methods).

The `getSignature` method requires students to implement conditional logic to return either just the last name (if the first name is empty) or the first letter of the first name followed by a dash and the last name. Successful responses used a variety of approaches to check whether the first name was empty, including comparing it to the empty string (`""`) or checking its length. Most responses correctly extracted the first character of the first name using `substring(0,1)` (or sometimes `charAt(0)`, which is outside the subset of Java covered in the Course and Exam Description (CED) but represents a valid approach to the problem that is eligible for full credit). Some responses failed to handle the empty first name case correctly or returned

the entire first name instead of just the first letter, but a solid majority of responses were successful at building the required string.

The `addSignature` method requires more complex string manipulation and conditional logic. Students needed to determine whether the signature (as returned by `getSignature`) was absent, at the end, or at the beginning of the input string parameter. Both the logic to determine the appropriate case and the resultant string-building proved to be particularly challenging.

Most responses made some correct use of conditional statements to partially distinguish the three cases, but only about a third of them fully and correctly distinguished all three cases. Common errors included failing to detect the signature at the start or end correctly, either via math errors in computing substring indices or in mishandling of an `indexOf` return value.

Each of the three cases then requires accessing or building the appropriate revised string and returning it. This also was challenging, with only a third of responses doing so correctly. Here, common errors included incorrectly removing or appending the signature or mishandling cases where the signature was not present. Other responses incorrectly added extra characters or delimiters when repositioning the signature. A few responses failed to return the constructed string properly, either returning `null` or printing the string instead of returning it.

What common student misconceptions or gaps in knowledge were seen in the responses to this question?

<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to create objects of a class and call methods.</i>	<i>Responses that Demonstrate Understanding</i>
Some responses incorrectly constructed the string for the first initial of the first name. <code>firstName.substring(0)</code> OR <code>firstName[0]</code> OR <code>firstName(0)</code>	<code>firstName.substring(0,1)</code>

<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to define a new type by creating a class.</i>	<i>Responses that Demonstrate Understanding</i>
Some responses used incorrect syntax to declare the class, occasionally combining the class header and constructor header. <pre>public class SignedText()</pre> OR <pre>public class SignedText(String f, String l)</pre>	<pre>public class SignedText</pre>
Some responses failed to declare the instance variables with private access or included a different modifier. <pre>public String firstName;</pre> OR <pre>String lastName;</pre>	<pre>private String firstName; private String lastName;</pre>
Some responses declared and initialized variables on the same line, outside the constructor, with non-constant values that didn't exist (in that scope). <pre>private String firstName = fst; private String lastName = lst;</pre> <pre>public SignedText(String fst, String lst) { }</pre>	<pre>private String firstName; private String lastName;</pre> <pre>public SignedText(String fst, String lst) { firstName = fst; lastName = lst; }</pre>
Some responses declared and assigned local variables inside the constructor instead of initializing instance variables. <pre>public SignedText(String fst, String lst) { String firstName = fst; String lastName = lst; }</pre>	<pre>private String firstName; private String lastName;</pre> <pre>public SignedText(String fst, String lst) { firstName = fst; lastName = lst; }</pre>

Some used the same name for instance variables and constructor parameters (without adjusting for it). <pre>public SignedText(String firstName, String lastName) { firstName = firstName; lastName = lastName; }</pre>	<pre>public SignedText(String fst, String lst) { firstName = fst; lastName = lst; }</pre> OR (outside the CED but commonly taught): <pre>public SignedText(String firstName, String lastName) { this.firstName = firstName; this.lastName = lastName; }</pre>
<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to satisfy method specifications using expressions and conditional statements.</i>	<i>Responses that Demonstrate Understanding</i>
Some responses incorrectly identified the empty string. <pre>firstName.equals(" ") // single space</pre> OR <pre>firstName.equals(null)</pre>	<pre>firstName.equals("")</pre>
Some responses used <code>==</code> instead of <code>equals</code> for string comparison. <pre>firstName == "" text.substring(0, sigLength) == sig</pre>	<pre>firstName.equals("") text.substring(0, sigLength).equals(sig)</pre>
Some responses failed to test all three cases (no match, match start, match end). <pre>if (text.indexOf(sig) >= 0) { ... } else { ... }</pre>	<pre>if (text.indexOf(sig) < 0) { ... } else if (text.indexOf(sig) == 0) { ... } else { ... }</pre>

Some responses incorrectly removed the signature substring from the beginning or end or both. <pre>int slen = sig.length(); ... text.substring(slen + 1) ... text.substring(0, slen)</pre>	<pre>int slen = sig.length(); ... text.substring(slen) ... text.substring(0, text.length() - slen)</pre>
Some responses confused the parameter string and the signature string, for example using the signature variable where the parameter string should be used, or vice versa. <pre>sig.substring(0, sigLength).equals(text)</pre> OR <pre>text.substring(0, textLength).equals(sig)</pre> OR <pre>sig.substring(text.length()) + text</pre> OR <pre>text.substring(sig.length()) + text</pre>	<pre>text.substring(0, sigLength).equals(sig) text.substring(sigLength) + sig</pre>

Based on your experience at the AP® Reading with student responses, what advice would you offer teachers to help them improve student performance on the exam?

- Even if your students sometimes use an IDE to fill out “boilerplate” code like class and method headers, make sure to practice writing out all those pieces without help from tools.
- More practice with string manipulation is always a good idea, and make sure to talk about the inclusive-exclusive pattern used for the arguments to `substring`, and practice with it to know when a `+1` or `-1` is or is not needed.

What resources would you recommend to teachers to better prepare their students for the content and skill(s) required on this question?

- Progress checks from units 2 and 3 (formerly 3 and 5) would be helpful to scaffold students’ understanding for the Class Design free-response questions.
- The following Topic Questions can be found in AP Classroom to support this Class Design free-response question:
 - Write program code to create objects of a class and call methods. Topics 1.9, 1.14, and 1.15 (formerly 2.3, 2.4, 2.5, and 2.7)
 - Write program code to define a new type by creating a class. Topics in unit 3 (formerly 5).
 - Write program code to satisfy method specifications using expressions and conditional statements. Topics 2.2 and 2.3 (formerly 3.2, 3.3, and 3.4).

Question 3

Task: Array / ArrayList

Topic: Tournament Competitors

	Max Points:	Mean Score:
Point 1: Access all in array	1	0.56
Point 2: Maintain incremented value	1	0.52
Point 3: Construct object	1	0.45
Point 4: Accumulate list (algorithm)	1	0.45
Point 5: Construct new list	1	0.51
Point 6: Index from both ends (algorithm)	1	0.51
Point 7: Determine even/odd case	1	0.55
Point 8: Access list, construct object, add to list	1	0.40
Point 9: Populate result list (algorithm)	1	0.19

Overall Mean Score: 4.12

What were the responses to this question expected to demonstrate?

This question tested the student's ability to:

- Write program code to create objects of a class and call methods (Skill 3.A).
- Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements (Skill 3.C).
- Write program code to create, traverse, and manipulate statements in 1D array and ArrayList objects (Skill 3.D).

Students were asked to pair competitors in a tournament into one-on-one matches for one round of the tournament. The students were given three classes: `Competitor`, `Match`, and `Round`. The `Competitor` class defined a single competitor in the tournament, which was created with a name and rank. The `Match` class represented the pair of competitors that would compete in the match in the tournament. The `Round` class represented the single round in the tournament and had a list of paired competitors. Successful completion of this question required accessing both arrays and ArrayLists, constructing objects, traversing in an unusual order, and adjusting an algorithm to separately handle even/odd cases.

In part (a), students were asked to write the constructor for the `Round` class. The constructor should initialize the `competitorList` with an ArrayList containing one `Competitor` object for each name given in the parameter array called `names`. Doing so requires a traversal capable of accessing all names in the array, creating a variable that would track a rank starting at 1, incrementing the rank by one for each iteration of the loop, then creating a `Competitor` object with the name and computed rank. Once each competitor object is created it should be added to the instance variable `competitorList`.

In part (b), students were asked to write the method `buildMatches` in the `Round` class. This method should return a list of `Match` objects consisting of competitors from the `competitorList` variable. The highest-ranked competitors are generally paired with the lowest-ranked competitors, so the traversal requires some means of tracking inward from each end of the list. Furthermore, handling the case with an odd number of competitors requires skipping or removing the highest-ranked competitor, without skipping or duplicating any others or accessing values out of bounds (or destroying the data), as well as detecting the odd/even case

in the first place. In addition to the interesting traversal pattern, the method requires a number of standard `ArrayList` skills: accessing at a specified index, constructing a new one, adding values into it, and returning it once the algorithm is complete.

How well did the responses address the course content related to this question? How well did the responses integrate the skill(s) required on this question?

Write program code to create objects of a class and call methods (Skill 3.A).

In both parts, a correct solution requires constructing an instance of a provided class; in part (a) this is a `Competitor` object, and in part (b) it is a `Match` object. Roughly half of the responses were basically successful at creating the objects. Of the responses that failed to create either a `Competitor` or a `Match` or both, there were two common reasons why: some responses had errors in the construction expression itself (e.g. by omitting the keyword `new`). Others omitted mention of the container classes entirely, often calling the `add` method on the `ArrayList` directly with what should be the contents of the object to be added.

Among the responses that did make a call to explicitly construct a `Competitor` or `Match` object, most had valid and correct parameters.

In addition, the problem requires instantiation of `ArrayList` objects and various method calls associated with them, considered under Skill 3.D below.

Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements (Skill 3.C).

This question tested this skill with some of the smaller logical pieces that were needed within the larger array and list algorithms. In part (a), students needed to initialize and maintain a counter to use as the rank—either as a separate variable starting at 1 or as a variable starting at 0 (possibly also used as an index) that explicitly gets added to 1 for use as a rank. A bare majority of responses were successful in doing so. Others either used it as a rank but started it at 0 (without adding 1 when using it as a rank), or didn't have the concept of rank anywhere in the response, in some cases not maintaining an `int` variable at all.

In part (b), students needed to identify and separately handle the even/odd case, generally with an `if` statement and a condition with a mathematical evenness check. Most responses were able to do so, although some had trouble with the use of the remainder (`%`) operator or how to process its result.

Write program code to create, traverse, and manipulate statements in 1D array and `ArrayList` objects (Skill 3.D).

This is the primary skill tested by this question, and it is tested throughout both parts. In part (a), the students had to use an iterative construct such as a `for` loop to access all the elements in the parameter array called `names`. Then, for each name, it must create and add a `Competitor` object to the `ArrayList` instance variable called `competitorList`. Most responses were successful with traversing the array, but only about half of them put the algorithm together and populated the list. A common difficulty was calling the `add` method directly with a name and rank, instead of first constructing a `Competitor` object and then adding that.

In part (b), this skill was tested in two distinct ways. First, the question tested the syntax and mechanics of working with `ArrayList` values, requiring the student to construct a new list, access values by index in an existing list, and add constructed values to the new list. Most responses successfully created the `ArrayList`

needed to store the `Match` objects. The complexity of the get-construct-and-add process was more difficult, though; many responses used `[]` to access the list, others omitted the “get” part entirely and tried to construct a `Match` object using indices rather than the `Competitor` objects *at* those indices, and still others skipped constructing a `Match` entirely and just provided both `Competitor` objects as arguments of `add`.

The second way this skill was tested in part (b) was through reasoning about the complicated list traversal required to identify all the corresponding pairs of competitors. One index needs to start at or near the beginning of the list; the other needs to be either a separately-maintained variable that starts at the end of the list or a value computed arithmetically from the first index. About half of the responses got at least as far as maintaining indices that generally march from both ends of the list towards the middle, but few responses managed the full traversal correctly. Typical problems included off-by-one arithmetic when computing the second index, miscalculating the “middle” stopping point, or accounting incorrectly for the omitted first element in the odd case. Another very common category of error arose when the response’s strategy for the odd case involved using `remove` to reduce it to the even case—a valid strategy, but to avoid destruction of data, it requires either operating on a copy of the original list or replacing the removed element after other processing is done. Most responses following the “remove first” strategy either made no copy at all or did the copying incorrectly, and very few of them even attempted to replace the removed element.

What common student misconceptions or gaps in knowledge were seen in the responses to this question?

Note that in all examples in this table, `cL` is a stand-in for `competitorList`, and `mL` is the name of the locally-declared `ArrayList<Match>` that is being populated in part (b). Where needed, assume the provided code is preceded by

```
ArrayList<Competitor> cL = competitorList;  
ArrayList<Match> mL = new ArrayList<Match>();
```

<i>Common Misconceptions/Knowledge Gaps</i>	<i>Responses that Demonstrate Understanding</i>
<i>Write program code to create objects of a class and call methods.</i>	
Many responses constructed <code>Competitor</code> objects incorrectly, or failed to do so at all. <code>cL.add(name, rank);</code> OR <code>cL.add(Competitor(name, rank));</code>	<code>cL.add(new Competitor(name, rank));</code>
In some cases, a response constructed a <code>String</code> instead of a <code>Competitor</code> object. <code>cL.add(n + " rank: " + rank);</code>	<code>cL.add(new Competitor(n, rank));</code>

Many responses constructed <code>Match</code> objects incorrectly, or failed to do so at all. <pre>mL.add(cL.get(j), cL.get(k));</pre> OR <pre>mL.add(new Match(j, k));</pre> OR <pre>mL.add(Match(cL.get(j), cL.get(k)));</pre>	<pre>mL.add(new Match(cL.get(j), cL.get(k)));</pre>
<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements.</i>	<i>Responses that Demonstrate Understanding</i>
Some responses failed to maintain the rank due to declaring the variable inside the loop, so that it resets each time through a loop. <pre>for (String name : names) { int rank = 1; cL.add(new Competitor(name, rank)); rank++; }</pre>	<pre>int rank = 1; for (String name : names) { cL.add(new Competitor(name, rank)); rank++; }</pre>
Some responses incorrectly stored a variable for later use due to a scope error, using a variable outside the block where the variable was declared. <pre>if (cL.size() % 2 == 0) { int start = 0; } else { int start = 1; } for (int i = start; ...) {...}</pre>	<pre>int start; if (cL.size() % 2 == 0) { start = 0; } else { start = 1; } for (int i = start; ...) {...}</pre>

Some responses omitted an <code>else</code> after the <code>if</code> block, causing the list to be processed twice in one of the cases. <pre> if (cL.size() % 2 == 0) { for (int i = 0; ...) { ... } } for (int i = 1; ...) { ... } </pre>	<pre> if (cL.size() % 2 == 0) { for (int i = 0; ...) { ... } } else { for (int i = 1; ...) { ... } } </pre>
Some responses redeclared the method parameter as a local variable. <pre> public Round (String[] names) { String[] names = ...; for (String name : names) {...} } </pre>	<pre> public Round (String[] names) { for (String name : names) {...} } </pre>
<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to create, traverse, and manipulate statements in 1D array and <code>ArrayList</code> objects.</i>	<i>Responses that Demonstrate Understanding</i>
Some responses use the enhanced <code>for</code> loop syntax to access the <code>names</code> array, but did so incorrectly. <pre> for (name : names) {...} </pre> OR <pre> for (int name : names) {...} </pre> OR <pre> for (String name: names) { String x = names[name]; } </pre> OR <pre> for (int i : names) { String x = names[i]; } </pre>	<pre> for (String name : names) {...} </pre>

Some responses constructed <code>ArrayList</code> objects incorrectly, or failed to do so at all. <pre>ArrayList<Match> matches; // no init</pre> OR <pre>ArrayList matches = ArrayList<>;</pre>	<pre>ArrayList<Match> matches = new ArrayList<Match>();</pre>
Some responses constructed an <code>ArrayList</code> correctly but assigned it to a local variable instead of the required instance variable. <pre>ArrayList<Competitor> competitorList = new ArrayList<Competitor>();</pre>	<pre>competitorList = new ArrayList<Competitor>();</pre>
Most responses handling the odd case with a “remove first” strategy operated on <code>competitorList</code> directly without restoring it. <pre>if (cL.size() % 2 != 0) { cL.remove(0); } ...</pre>	<pre>Competitor ogFirst; if(cL.size() % 2 != 0) { ogFirst = cL.remove(0); } ... cL.add(0, ogFirst);</pre>
Many of the responses following a “remove first” strategy operated on an alias instead of creating a copy. <pre>ArrayList<Competitor> temp = cL; temp.remove(0);</pre>	<pre>ArrayList<Competitor> temp = new ArrayList<Competitor>(); for (int i = 1; i < cL.size(); i++) { temp.add(cL.get(i)); }</pre> OR (outside the CED but commonly taught): <pre>ArrayList<Competitor> temp = new ArrayList<Competitor>(cL); temp.remove(0);</pre>

Some responses made incorrect calls to the <code>get</code> or <code>add</code> methods (or omitted them). <pre>mL.add(new Match(cL(i), cL(cL.size() - i)));</pre> OR <pre>mL.add(Match(cL[i], cL[cL.size() - 1]));</pre> OR <pre>add(i, cL.size() - i);</pre> OR <pre>add(Match(i, cL.size() - 1));</pre>	<pre>mL.add(new Match(cL.get(i), cL.get(cL.size() - i)));</pre>
Some responses confused the <code>set</code> method with the <code>add</code> method. <pre>Match m = new Match(cL.get(l), cL.get(h)); matches.set(0, m);</pre>	<pre>Match m = new Match(cL.get(l), cL.get(h)); matches.add(0, m);</pre>
Some responses use an incorrect condition to end the loop at the midpoint of the list for the even case. <pre>for (int i = 0; i < cL.size() / 2 - 1; i++)</pre>	<pre>for (int i = 0; i < cL.size() / 2; i++)</pre>
Many responses used an incorrect condition statement to end the loop at the midpoint of the list for the odd case. <pre>for (int i = 1; i < cL.size() / 2; i++)</pre> OR <pre>for (int i = 1; i < cL.size() / 2 - 1; i++)</pre> OR <pre>for (int i = 1; i < (cL.size()-1) / 2; i++)</pre>	<pre>for (int i = 1; i <= cL.size() / 2; i++)</pre> OR <pre>for (int i = 1; i < cL.size() / 2 + 1; i++)</pre> OR <pre>for (int i = 1; i < (cL.size()+1) / 2; i++)</pre>

Many responses using a single variable with arithmetic to index from both ends of the list made a bounds error in the even case when the loop variable is 0.

```
int size = cL.size();
for (int i = 0; i < size / 2; i++)
{
    Competitor one = cL.get(i);
    Competitor two = cL.get(size - i);
    ...
}
```

```
int size = cL.size();
for (int i = 0; i < size / 2; i++)
{
    Competitor one = cL.get(i);
    Competitor two = cL.get(size - i - 1);
    ...
}
```

Based on your experience at the AP® Reading with student responses, what advice would you offer teachers to help them improve student performance on the exam?

- When defining a constructor, always make sure that all instance variables have been initialized, especially any that may hold objects requiring instantiation.
- Practice working with `ArrayList` variables that contain values of a provided class; accessing and building lists like that often require chaining method calls (e.g. `list.get(i).getValue()`) or nesting a constructor call inside a method call (e.g. `list.add(new Something(value, value))`).
- When unusual traversal patterns are in play, practice tracing draft solutions on small but concrete sample values (lists of four or five elements) to find errors before submitting.
 - TIP: I literally hold up my left hand to make “lists” of 3-5 elements that I can use my right index finger to count through and confirm where traversals start or end, what order they occur, and so on. Visuals help! Students can also do a more formal trace on paper, but counting-on-fingers is a great way to quickly check bounds logic.
- Encourage students to plan out their algorithm, and if it seems like it will require a large segment of nearly-duplicate code, pause a moment before proceeding to see if there’s a way to handle it without duplicate code. Sometimes it’s unavoidable (and, if no other ideas present themselves, better to write something than nothing!) but near-duplicate code, even if copy-and-paste are available, is an invitation for bugs.

What resources would you recommend to teachers to better prepare their students for the content and skill(s) required on this question?

- Progress checks from unit 4 (formerly 6 and 7) would be helpful to scaffold students’ understanding for the array/`ArrayList` free-response questions. (Note that under the new CED, Q3 will not include array content, though array content will appear elsewhere in the exam.)
- The following Topic Questions can be found in AP Classroom to support this array/`ArrayList` free-response question:
 - Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements. Topics 1.3, 1.4, 1.14, 2.3, 2.8, 3.5, and 3.6 (formerly 1.3, 2.5, 3.3, 4.2, and 5.6).
 - Write code to create, traverse, and manipulate elements in a 1D array or `ArrayList` objects. Topics 4.4, 4.5, 4.8, 4.9, and 4.10 (formerly 6.1, 6.2, 6.3, 7.1, 7.2, 7.3, and 7.4).

Question 4

Task: 2D Array

Topic: Number Pairing Puzzle

	Max Points:	Mean Score:
Point 1: Construct 2D array	1	0.33
Point 2: Traverse 2D array	1	0.57
Point 3: Random integer in range	1	0.43
Point 4: Assign elements (algorithm)	1	0.63
Point 5: Access all in 2D array	1	0.37
Point 6: Handle special case	1	0.12
Point 7: Compare values	1	0.54
Point 8: Assign element	1	0.56
Point 9: Update and return (algorithm)	1	0.40
Overall Mean Score:		3.96

What were the responses to this question expected to demonstrate?

This question tested the student's ability to:

- Write program code to call methods (Skill 3.A).
- Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements (Skill 3.C).
- Write program code to create, traverse, and manipulate elements in 2D array objects (Skill 3.E).

This question described a class with a gameboard, represented as an instance variable that is a two-dimensional (2D) array of `int` values. Students were asked to write code to construct, populate, and traverse the 2D array, to generate random integers in a specified range, and to logically combine several different conditional checks to ensure that only the specified updates occur.

In part (a), students were asked to write a constructor for `SumOrSameGame` that has two integer parameters, `numRows` and `numCols`, and constructs a new 2D array of `int` elements with `numRows` rows and `numCols` columns. The elements of the 2D array should then be filled with random `int` values in the range 1 through 9, inclusive.

In part (b), students were asked to write an instance method, `clearPair`, which takes two integer parameters, `row` and `col`, and searches the gameboard for an element of the 2D array in the row with index `row` or beyond to pair with the element at `row` and `col` such that the two elements either have the same value or their values sum to 10. If a pair is found, the element at `row` and `col` and its paired element should each be cleared (set to 0) and the method should return `true`; if no such pair exists, the method should return `false`.

How well did the responses address the course content related to this question? How well did the responses integrate the skill(s) required on this question?

Write program code to call methods (Skill 3.A).

Part (a) tested this skill by requiring the generation of a random number. Most responses provided a

syntactically correct call to `Math.random()`. Many of these responses were not, however, able to transform the result returned by `Math.random()` into an integer in the appropriate range from 1 through 9, inclusive. (There are other Java classes and methods, outside of the subset described in the Course and Exam Description (CED), that provide functionality for generating random values (such as the `Random` class), and a small number of responses used them successfully, earning credit. The exam never requires the use of methods outside the subset; but responses that use them incorrectly will not earn the associated points.)

Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements (Skill 3.C).

Part (b) tested this skill extensively with the logic for identifying a pair of distinct elements of the 2D array that meet the sum-or-same condition specified in the prompt, as well as for the algorithmic structure to return `true` if and only if such a pair was identified.

The vast majority of responses did not correctly guard against pairing an element with itself; this was the single most difficult point among all the FRQs. A large portion did not write any code that even attempted to do so, possibly indicating a gap between shallow understanding of the syntactic mechanics of traversal versus a deeper understanding of what happens in the traversal and what needs to happen to accomplish the task. Among responses that attempted to prevent the element at `puzzle[row][col]` from being paired with itself, many produced guards that excluded all elements in the entire row and column rather than just the individual element.

Most responses wrote appropriate code to check that two elements had the same value or that two elements' values added up to 10. Common errors were checking only one of the two conditions or logically combining them incorrectly, but some responses had logical errors in the condition checks themselves.

Most responses wrote appropriate code to return `true` when identifying a pair of elements that met the required sum-or-same conditions. The case to return `false`, however was incorrect in many responses, with a common algorithmic error being a too-early return if the first pair checked did not meet the conditions. Some responses simply neglected to return at all after all pairs had been considered. There were also some responses with an algorithmic error of clearing extra elements due to not returning immediately after handling the first identified pair to meet the conditions.

Write program code to create, traverse, and manipulate elements in 2D array objects (Skill 3.E).

In part (a), responses needed to construct and fill a new 2D array of the proper size given by the parameters and assign it to the specified instance variable. Most responses did not instantiate the 2D array, did so syntactically incorrectly, or assigned it to a local variable rather than the instance variable. Despite this, most responses did traverse the 2D array correctly with no bounds errors, and assigned a value at each element.

In part (b), responses needed to traverse only a portion of the 2D array, and in some cases modify its elements. A large majority of responses wrote code that attempted to traverse the 2D array, however most were not able to access exactly the specified portion—with many accessing every element and others over-constraining the traversal in a way that missed required elements. A prevalent syntactic error was in identifying the number of columns in the 2D array to use in the loop condition for the traversal over a row. When handling an element of the 2D array, most responses were able to set an element to 0—some responses omitted this entirely, however, an occasional syntactic error was to treat the 2D array instance variable as if it were an `ArrayList` and attempt to use instance methods such as `set` or `remove`.

What common student misconceptions or gaps in knowledge were seen in the responses to this question?

<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to call methods.</i>	<i>Responses that Demonstrate Understanding</i>
Many responses omitted the cast to <code>int</code>, or completed the cast before multiplying to scale the result. <code>Math.random() * 9 + 1</code> OR <code>(int) Math.random() * 9 + 1</code>	<code>(int) (Math.random() * 9) + 1</code> OR <code>(int) (Math.random() * 9 + 1)</code>
Many responses scaled the result from <code>Math.random()</code> by the wrong amount, or failed to shift the range to start at the right number. <code>(int) (Math.random() * 8) + 1</code> OR <code>(int) (Math.random() * 10)</code> OR <code>(int) (Math.random() * 9)</code>	<code>(int) (Math.random() * 9) + 1</code> OR <code>(int) (Math.random() * 9 + 1)</code>
<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements.</i>	<i>Responses that Demonstrate Understanding</i>
Most responses omitted the self-pairing guard entirely. Among the responses with a clear attempt at such a guard, the majority excluded the entire row and entire column instead of just the single “self” element. <code>if (r != row && c != col)</code> <code>{</code> <code> // do sum-or-same checks</code> <code>}</code>	<code>if (r != row c != col) {...}</code> OR <code>if (!(r == row && c == col)) {...}</code>

Some responses were logically incorrect in their check that two elements may be paired if <i>either</i> of the two conditions is satisfied. <pre> if (puzzle[r][c] == puzzle[row][col] && puzzle[r][c] + puzzle[row][col] == 10) { // clear pair } </pre>	<pre> if (puzzle[r][c] == puzzle[row][col] puzzle[r][c] + puzzle[row][col] == 10) { // clear pair } </pre> OR <pre> if (puzzle[r][c] == puzzle[row][col]) { // clear pair } if (puzzle[r][c] + puzzle[row][col] == 10) { // clear pair } </pre>
<i>Common Misconceptions/Knowledge Gaps</i> <i>Write program code to create, traverse, and manipulate elements in 2D array objects.</i>	<i>Responses that Demonstrate Understanding</i>
Many responses assigned the newly constructed 2D array to a local variable instead of the instance variable <code>puzzle</code>. <pre> int[][] arr2D = new int[numRows][numCols]; </pre>	<pre> puzzle = new int[numRows][numCols]; </pre> OR <pre> int[][] arr2D = new int[numRows][numCols]; ... puzzle = arr2D; </pre>
Many responses traversed the entire 2D array instead of only rows starting at index <code>row</code>. <pre> for (int r = 0; r < puzzle.length; r++) </pre>	<pre> for (int r = row; r < puzzle.length; r++) </pre>
Many responses traversed only columns at or to the right of the target element. <pre> for(int c = col; c < puzzle[r].length; c++) </pre>	<pre> for (int c = 0; c < puzzle[r].length; c++) </pre>

Some responses used the number of rows as a column bound or incorrectly accessed the number of columns in the 2D array.	
<pre>for (int c = 0; c < puzzle[].length; c++)</pre>	<pre>for(int c = 0; c < puzzle[r].length; c++)</pre>

Based on your experience at the AP® Reading with student responses, what advice would you offer teachers to help them improve student performance on the exam?

- Practice constructing and filling 2D arrays.
- Differentiate using a class's instance variables versus a constructor's or method's local variables.
- Practice creating a random number in a given range. Many students have clearly been coached to follow the generalized pattern:

```
(int) (Math.random() * (rangeMax - rangeMin + 1)) + rangeMin
```

This drilling is better than making students have to figure it out during the exam, but teachers can still aim for a deeper understanding rather than just being purely pattern-driven.
- Practice traversal over 2D arrays, including problems that require traversals over only a subset of the elements rather than the entire array or algorithms that require more sophisticated manipulation of the 2D array than a rote traversal.
- Differentiate identifying the *same element* (location) in an array versus identifying locations in the array that have the same *value*. Accessing the same location in a data structure will necessarily read the same value, whereas reading the same value out of a data structure twice does not necessarily mean that both came from the same location.

What resources would you recommend to teachers to better prepare their students for the content and skill(s) required on this question?

- Progress checks from unit 4 (formerly 8) would be helpful to scaffold students' understanding for the 2D array free-response questions.
- The following Topic Questions can be found in AP Classroom to support this 2D array free-response question:
 - Write program code to create objects of a class and call methods. Topics 1.11, 1.13, and 1.14 (formerly 2.2, 2.5 and 2.9).
 - Write program code to satisfy method specifications using expressions, conditional statements, and iterative statements. Topics 2.4, 2.5, 2.6, and 2.8 (formerly 3.1, 3.3, 3.4, 3.5, 3.6, and 4.2).
 - Write program code to create, traverse, and manipulate elements in 2D array objects. Topics 4.11, 4.12, and 4.13 (formerly 8.1 and 8.2).